5. Bottom-Up Parsing

- LR methods (Left-to-right, Rightmost derivation)
  - SLR, Canonical LR, LALR
- Other special cases:
  - Shift-reduce parsing
  - Operator-precedence parsing

Shift-Reduce Parsing

Grammar:

$S \rightarrow \mathbf{a}\ A\ B\ \mathbf{e}$

$A \rightarrow A\ \mathbf{b}\ \mathbf{c} \mid \mathbf{b}$

$B \rightarrow \mathbf{d}$

These match production's right-hand sides

Reducing a sentence:

$\mathbf{a}\ \underline{\mathbf{b}}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ \underline{A\ \mathbf{b}\ \mathbf{c}}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ A\ \underline{\mathbf{d}}\ \mathbf{e}$

$\underline{\mathbf{a}\ A\ B\ \mathbf{e}}$

S

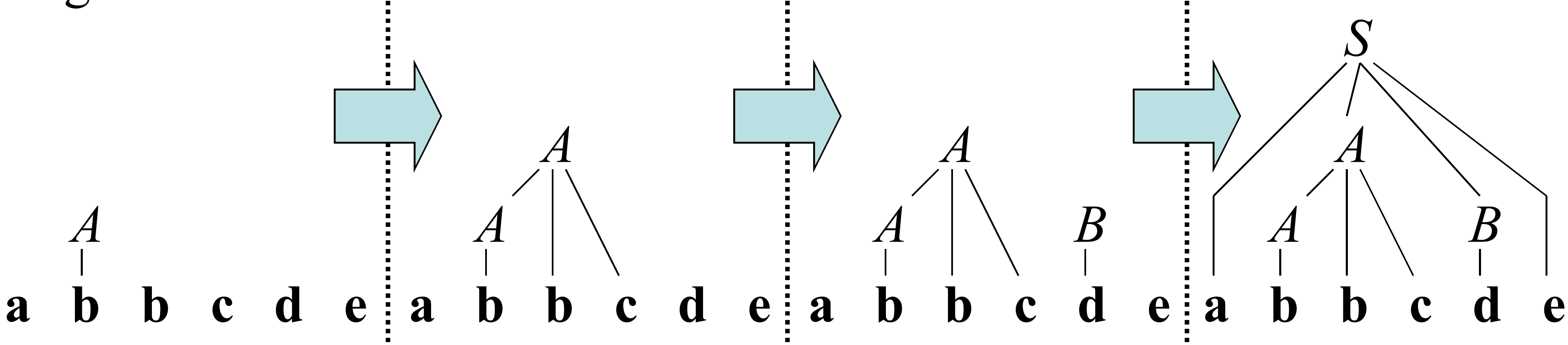
Shift-reduce corresponds to a rightmost derivation:

$S \Rightarrow_{rm} \mathbf{a}\ A\ B\ \mathbf{e}$

$\Rightarrow_{rm} \mathbf{a}\ A\ \mathbf{d}\ \mathbf{e}$

$\Rightarrow_{rm} \mathbf{a}\ A\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$

$\Rightarrow_{rm} \mathbf{a}\ \mathbf{b}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$



Handles

A *handle* is a substring of grammar symbols in a *right-sentential form* that matches a right-hand side of a production

Grammar:

$S \to \mathbf{a}\ A\ B\ \mathbf{e}$

$A \to A\ \mathbf{b}\ \mathbf{c} \mid \mathbf{b}$

$B \to \mathbf{d}$

$\mathbf{a}\ \underline{\mathbf{b}}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ \underline{A\ \mathbf{b}\ \mathbf{c}}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ A\ \underline{\mathbf{d}}\ \mathbf{e}$

$\underline{\mathbf{a}\ A\ B\ \mathbf{e}}$

S

Handle

$\mathbf{a}\ \underline{\mathbf{b}}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ A\ \underline{\mathbf{b}}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ A\ A\ \mathbf{e}$

… ?

NOT a handle, because further reductions will fail (result is not a sentential form)

Stack Implementation of Shift-Reduce Parsing

Grammar:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow \mathbf{id}$

Stack	Input	Action
$	**id+id*id$**	shift
**$**$\underline{\mathbf{id}}$	**+id*id$**	reduce $E \rightarrow \mathbf{id}$
**$**$E$	**+id*id$**	shift
**$**$E$+	**id*id$**	shift
**$**$E$+$\underline{\mathbf{id}}$	***id$**	reduce $E \rightarrow \mathbf{id}$
**$**$E$+$E$	***id$**	shift (or reduce?)
**$**$E$+$E$*	**id$**	shift
**$**$E$+$E$*$\underline{\mathbf{id}}$	**$**	reduce $E \rightarrow \mathbf{id}$
**$**$E$+$\underline{E*E}$	**$**	reduce $E \rightarrow E * E$
**$**$\underline{E+E}$	**$**	reduce $E \rightarrow E + E$
**$**$E$	**$**	accept

How to resolve conflicts?

Found handles to reduce

Conflicts

- *Shift-reduce* and *reduce-reduce* conflicts are caused by
  - The limitations of the LR parsing method (even when the grammar is unambiguous)
  - Ambiguity of the grammar

Shift-Reduce Parsing: Shift-Reduce Conflicts

Ambiguous grammar:

$S \rightarrow$ **if** E **then** S
    | **if** E **then** S **else** S
    | **other**

Stack	Input	Action
$…	…**$**	…
$…<u>**if** *E* **then** *S*</u>	**else**…**$**	shift or reduce?

Resolve in favor of shift, so **else** matches closest **if**

Shift-Reduce Parsing: Reduce-Reduce Conflicts

Grammar:

$C \rightarrow A\ B$

$A \rightarrow \mathbf{a}$

$B \rightarrow \mathbf{a}$

Stack	Input	Action
$	**aa$**	shift
$<u>a</u>	**a$**	reduce $A \rightarrow \mathbf{a}$ <u>or</u> $B \rightarrow \mathbf{a}$?

Resolve in favor of reducing $A \rightarrow \mathbf{a}$, otherwise we're stuck!

6. LR Parsing: Simple LR

- LR(k) parsing
  - From left to right scanning of the input
  - Rightmost derivation in reverse
  - k lookahead symbols, only consider k=0, or 1
- Why LR Parsers
  - Can recognize virtually all programming language constructs
  - the most general nonbacktracking shift-reduce parsing method
  - Can detect a syntactic error as soon as possible
  - Powerful than LL parsing methods

LR(0) Items of a Grammar

- An *LR*(0) *item* of a grammar G is a production of G with a • at some position of the right-hand side
- Thus, a production

  $A \rightarrow X\ Y\ Z$

  has four items:

  $[A \rightarrow \bullet\ X\ Y\ Z]$
  $[A \rightarrow X \bullet\ Y\ Z]$
  $[A \rightarrow X\ Y \bullet\ Z]$
  $[A \rightarrow X\ Y\ Z\ \bullet]$
- Note that production $A \rightarrow \varepsilon$ has one item $[A \rightarrow \bullet]$

The *closure* Operation for LR(0) Items

1. Start with $closure(I) = I$
2. If $[A \rightarrow \alpha \bullet B\beta] \in closure(I)$ then for each production $B \rightarrow \gamma$ in the grammar, add the item $[B \rightarrow \bullet\gamma]$ to I if not already in I
3. Repeat 2 until no new items can be added

The ***closure*** Operation Example

$closure(\{[E' \to \bullet E]\}) =$

$\{\ [E' \to \bullet\ E]\ \}$

$\{\ [E' \to \bullet\ E]$
$[E \to \bullet\ E + T]$
$[E \to \bullet\ T]\ \}$

Add $[E \to \bullet \gamma]$

$\{\ [E' \to \bullet\ E]$
$[E \to \bullet\ E + T]$
$[E \to \bullet\ T]$
$[T \to \bullet\ T * F]$
$[T \to \bullet\ F]\ \}$

Add $[T \to \bullet \gamma]$

$\{\ [E' \to \bullet\ E]$
$[E \to \bullet\ E + T]$
$[E \to \bullet\ T]$
$[T \to \bullet\ T * F]$
$[T \to \bullet\ F]$
$[F \to \bullet\ \mathbf{(}\ E\ \mathbf{)}]$
$[F \to \bullet\ \mathbf{id}]\ \}$

Add $[F \to \bullet \gamma]$

Grammar:

$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to \mathbf{(}\ E\ \mathbf{)}$
$F \to \mathbf{id}$

The ***goto*** Operation for LR(0) Items

1. For each item $[A \rightarrow \alpha \bullet X\beta] \in I$, add the set of items ***closure***$(\{[A \rightarrow \alpha X \bullet \beta]\})$ to ***goto***(I,X) if not already there
2. Repeat step 1 until no more items can be added to ***goto***(I,X)

- Intuitively, the ***goto*** function is used to define the transitions in the LR(0) automaton for a grammar.
- The states of the automaton correspond to sets of items, and ***goto***(I, X) specifies the transition from the state for I under input X.

The ***goto*** Operation Example 1

Suppose

$$
\begin{aligned}
I = \{\ & [E' \to \bullet\, E] \\
& [E \to \bullet\, E + T] \\
& [E \to \bullet\, T] \\
& [T \to \bullet\, T * F] \\
& [T \to \bullet\, F] \\
& [F \to \bullet\, \mathbf{(}\ E\ \mathbf{)}] \\
& [F \to \bullet\, \mathbf{id}]\ \}
\end{aligned}
$$

Then $goto(I, E)$

$= closure(\{[E' \to E\, \bullet, E \to E\, \bullet + T]\})$

$= \{\ [E' \to E\, \bullet], [E \to E\, \bullet + T]\ \}$

Grammar:

$E \to E + T \mid T$

$T \to T * F \mid F$

$F \to \mathbf{(}\ E\ \mathbf{)}$

$F \to \mathbf{id}$

The ***goto*** Operation Example 2

Suppose $I = \{\ [E' \to E\ \bullet],\ [E \to E\ \bullet + T]\ \}$

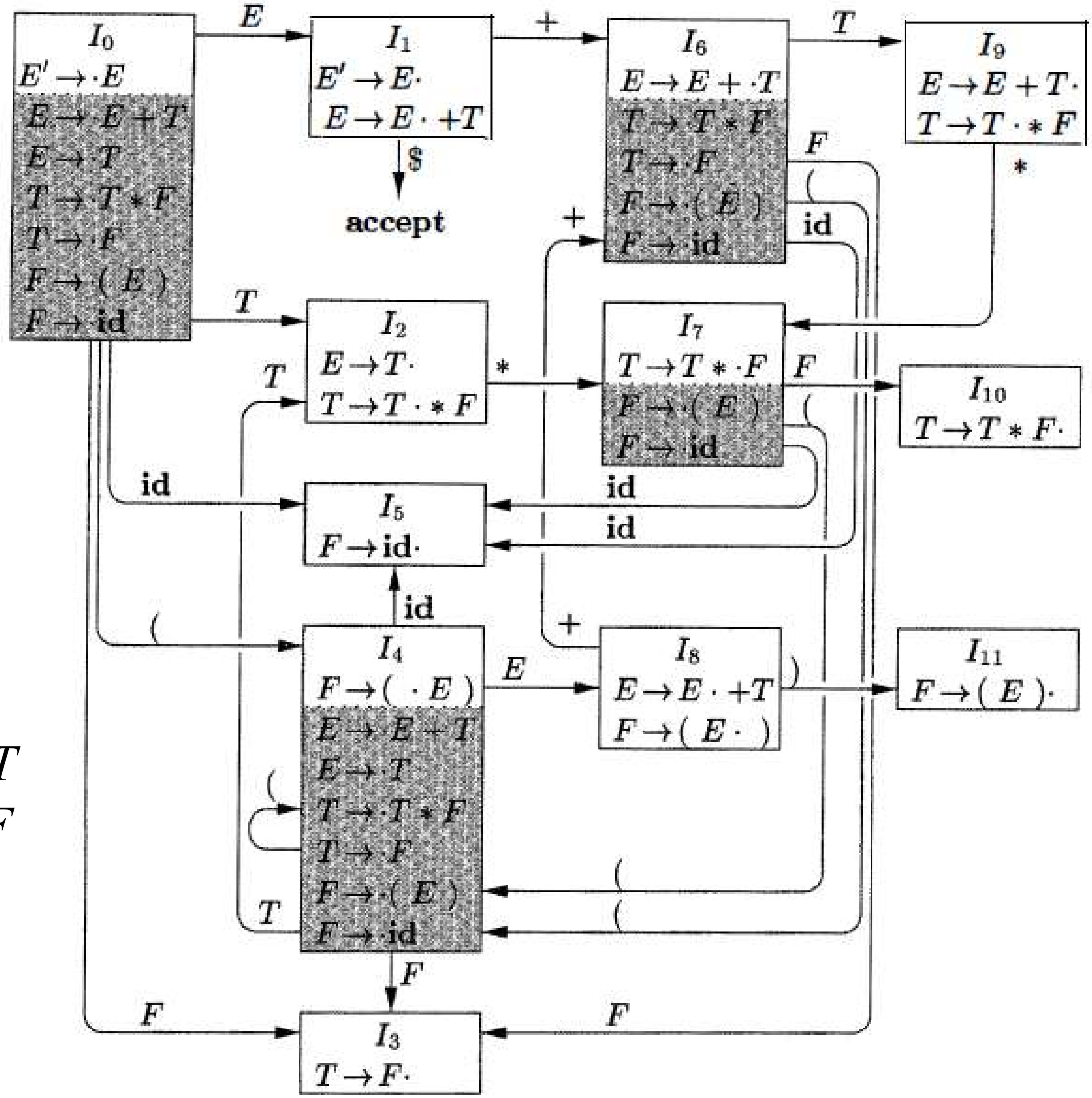
Then $goto(I,+) = closure(\{[E \to E + \bullet\ T]\}) = \{\ [E \to E + \bullet\ T]$
$[T \to \bullet\ T * F]$
$[T \to \bullet\ F]$
$[F \to \bullet\ \mathbf{(}\ E\ \mathbf{)}]$
$[F \to \bullet\ \mathbf{id}]\ \}$

Grammar:
$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to \mathbf{(}\ E\ \mathbf{)}$
$F \to \mathbf{id}$

Constructing the Canonical LR(0) Collection of a Grammar

1. The grammar is augmented with a new start symbol S' and production $S' \rightarrow S$
2. Initially, set $C = \{$ ***closure***$(\{[S' \rightarrow \bullet S]\})$ $\}$ (this is the start state of the DFA)
3. For each set of items $I \in C$ and each grammar symbol $X \in (N \cup T)$ such that ***goto***$(I,X) \notin C$ and ***goto***$(I,X) \neq \varnothing$, add the set of items ***goto***(I,X) to C
4. Repeat 3 until no more sets can be added to C

LR(0) Automaton for



Grammar:

$E \to E + T \mid T$

$T \to T * F \mid F$

$F \to (E)$

$F \to \mathbf{id}$

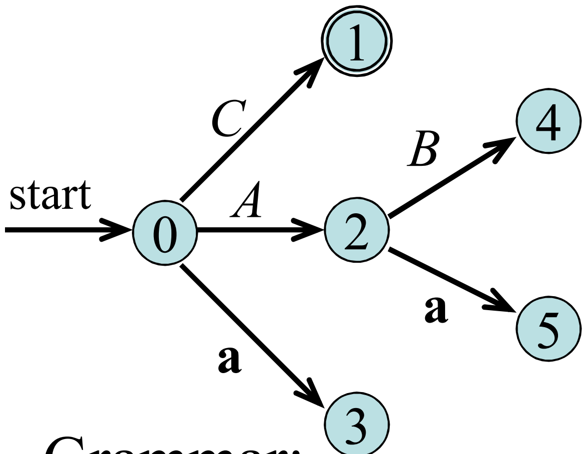
Use of the LR(0) Automaton

The following Figure shows the actions of a shift-reduce parser on input **id * id**, using the LR(0) automaton shown on previous slide.

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	$\$$	$\mathbf{id} * \mathbf{id}\ \$$	shift to 5
(2)	0 5	$\$\ \mathbf{id}$	$* \mathbf{id}\ \$$	reduce by $F \to \mathbf{id}$
(3)	0 3	$\$\ F$	$* \mathbf{id}\ \$$	reduce by $T \to F$
(4)	0 2	$\$\ T$	$* \mathbf{id}\ \$$	shift to 7
(5)	0 2 7	$\$\ T *$	$\mathbf{id}\ \$$	shift to 5
(6)	0 2 7 5	$\$\ T * \mathbf{id}$	$\$$	reduce by $F \to \mathbf{id}$
(7)	0 2 7 10	$\$\ T * F$	$\$$	reduce by $T \to T * F$
(8)	0 2	$\$\ T$	$\$$	reduce by $E \to T$
(9)	0 1	$\$\ E$	$\$$	accept

LR(k) Parsers: Use a DFA for Shift/Reduce Decisions

Use of the LR(0) Automaton



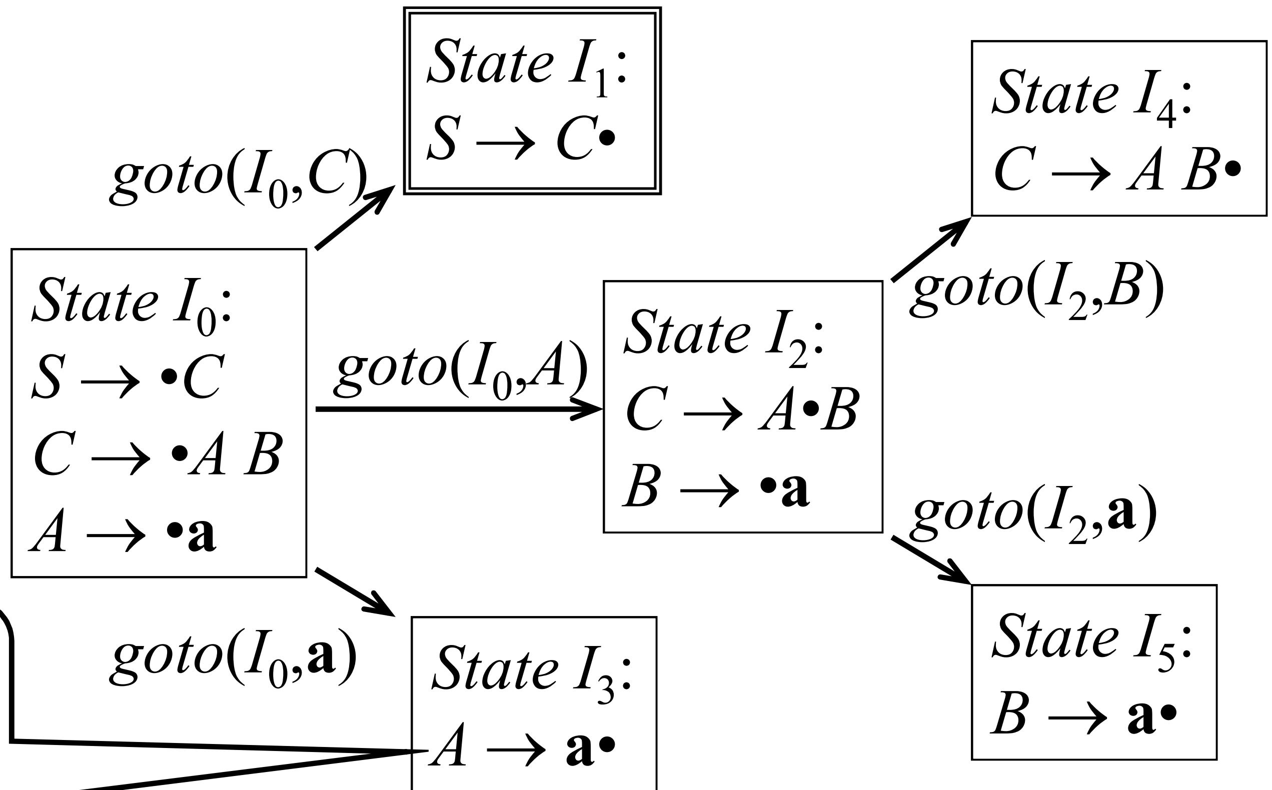
Grammar:

$S \to C$

$C \to A\ B$

$A \to \mathbf{a}$

$B \to \mathbf{a}$



Can only reduce $A \to \mathbf{a}$ (not $B \to \mathbf{a}$)

DFA for Shift/Reduce Decisions

The states of the DFA are used to determine if a handle is on top of the stack

Grammar:

$S \to C$

$C \to A\ B$

$A \to \mathbf{a}$

$B \to \mathbf{a}$

Stack	Symbols	Input	Action
0	**$**	**aa$**	shift to 3
0 3	**$a**	**a$**	reduce $A \to \mathbf{a}$
0 2	$A	**a$**	shift to 5
0 2 5	$A**a**	**$**	reduce $B \to \mathbf{a}$
0 2 4	$AB	**$**	reduce $C \to AB$
0 1	$C	**$**	accept ($S \to C$)

State I_0:

$S \to \bullet C$

$C \to \bullet A\ B$

$A \to \bullet\mathbf{a}$

$goto(I_0,\mathbf{a})$ $\longrightarrow$

State I_3:

$A \to \mathbf{a}\bullet$

DFA for Shift/Reduce Decisions

The states of the DFA are used to determine if a handle is on top of the stack

Grammar:

$S \to C$

$C \to A\ B$

$A \to \mathbf{a}$

$B \to \mathbf{a}$

Stack	Symbols	Input	Action
0	**$**	**aa$**	shift to 3
0 3	**$a**	**a$**	reduce $A \to \mathbf{a}$
0 2	$A	**a$**	shift to 5
0 2 5	$A**a**	**$**	reduce $B \to \mathbf{a}$
0 2 4	$AB	**$**	reduce $C \to AB$
0 1	$C	**$**	accept ($S \to C$)

State I_0:
$S \to \bullet C$
$C \to \bullet A\ B$
$A \to \bullet\mathbf{a}$

$goto(I_0,A)$

State I_2:
$C \to A \bullet B$
$B \to \bullet\mathbf{a}$

DFA for Shift/Reduce Decisions

The states of the DFA are used to determine if a handle is on top of the stack

Grammar:

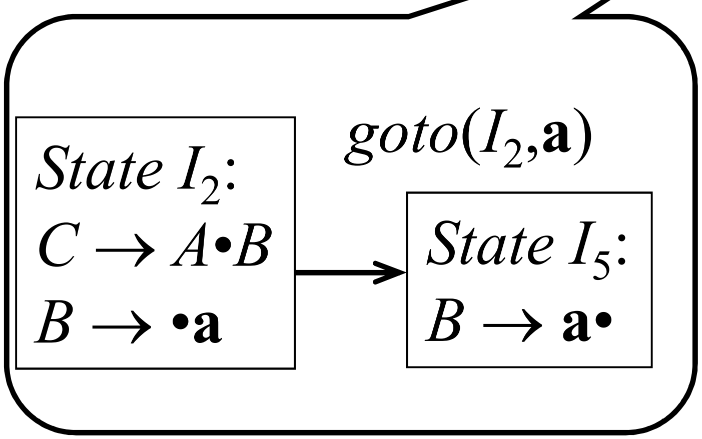
$S \rightarrow C$

$C \rightarrow A\ B$

$A \rightarrow \mathbf{a}$

$B \rightarrow \mathbf{a}$

Stack	Symbols	Input	Action
0	**$**	**aa$**	shift to 3
0 3	**$a**	**a$**	reduce $A \rightarrow \mathbf{a}$
0 2	$A	**a$**	shift to 5
0 2 5	$A**a**	**$**	reduce $B \rightarrow \mathbf{a}$
0 2 4	$AB	**$**	reduce $C \rightarrow AB$
0 1	$C	**$**	accept ($S \rightarrow C$)



DFA for Shift/Reduce Decisions

The states of the DFA are used to determine if a handle is on top of the stack

Grammar:
$S \rightarrow C$
$C \rightarrow A\ B$
$A \rightarrow \mathbf{a}$
$B \rightarrow \mathbf{a}$

Stack	Symbols	Input	Action
0	**$**	**aa$**	shift to 3
0 3	**$a**	**a$**	reduce $A \rightarrow \mathbf{a}$
0 2	$A	**a$**	shift to 5
0 2 5	$A**a**	**$**	reduce $B \rightarrow \mathbf{a}$
0 2 4	$AB	**$**	reduce $C \rightarrow AB$
0 1	$C	**$**	accept ($S \rightarrow C$)

State I_2:
$C \rightarrow A \bullet B$
$B \rightarrow \bullet \mathbf{a}$

$\xrightarrow{goto(I_2, B)}$

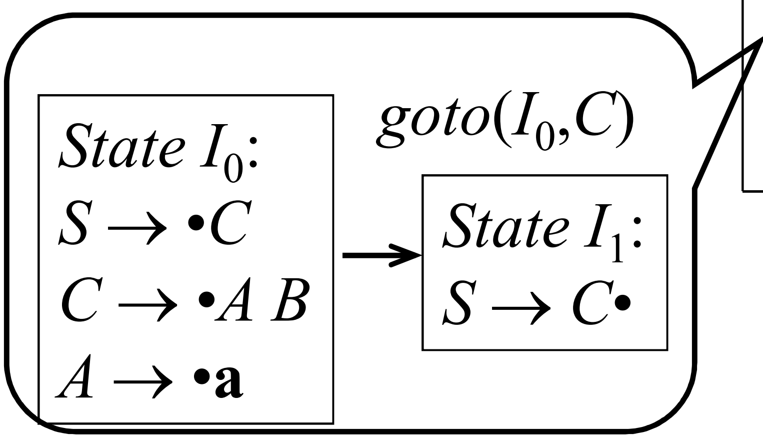
State I_4:
$C \rightarrow A\ B \bullet$

DFA for Shift/Reduce Decisions

The states of the DFA are used to determine if a handle is on top of the stack

Grammar:
$S \to C$
$C \to A\ B$
$A \to \mathbf{a}$
$B \to \mathbf{a}$

Stack	Symbols	Input	Action
0	**$**	**aa$**	shift to 3
0 3	**$a**	**a$**	reduce $A \to \mathbf{a}$
0 2	$A	**a$**	shift to 5
0 2 5	$A**a**	**$**	reduce $B \to \mathbf{a}$
0 2 4	$AB	**$**	reduce $C \to AB$
0 1	$C	**$**	accept ($S \to C$)



DFA for Shift/Reduce Decisions

The states of the DFA are used to determine if a handle is on top of the stack

Grammar:

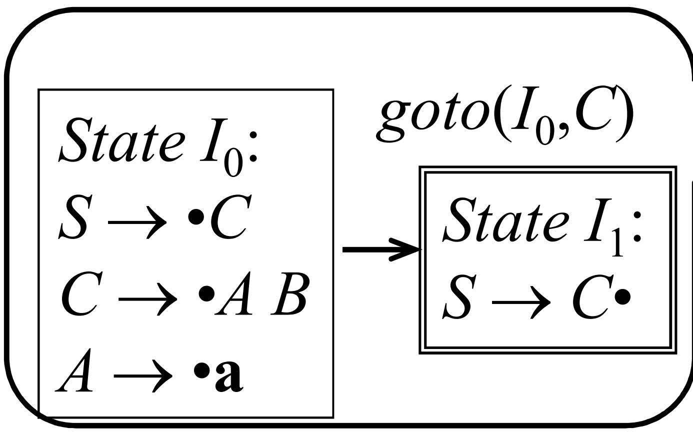
$S \to C$

$C \to A\ B$

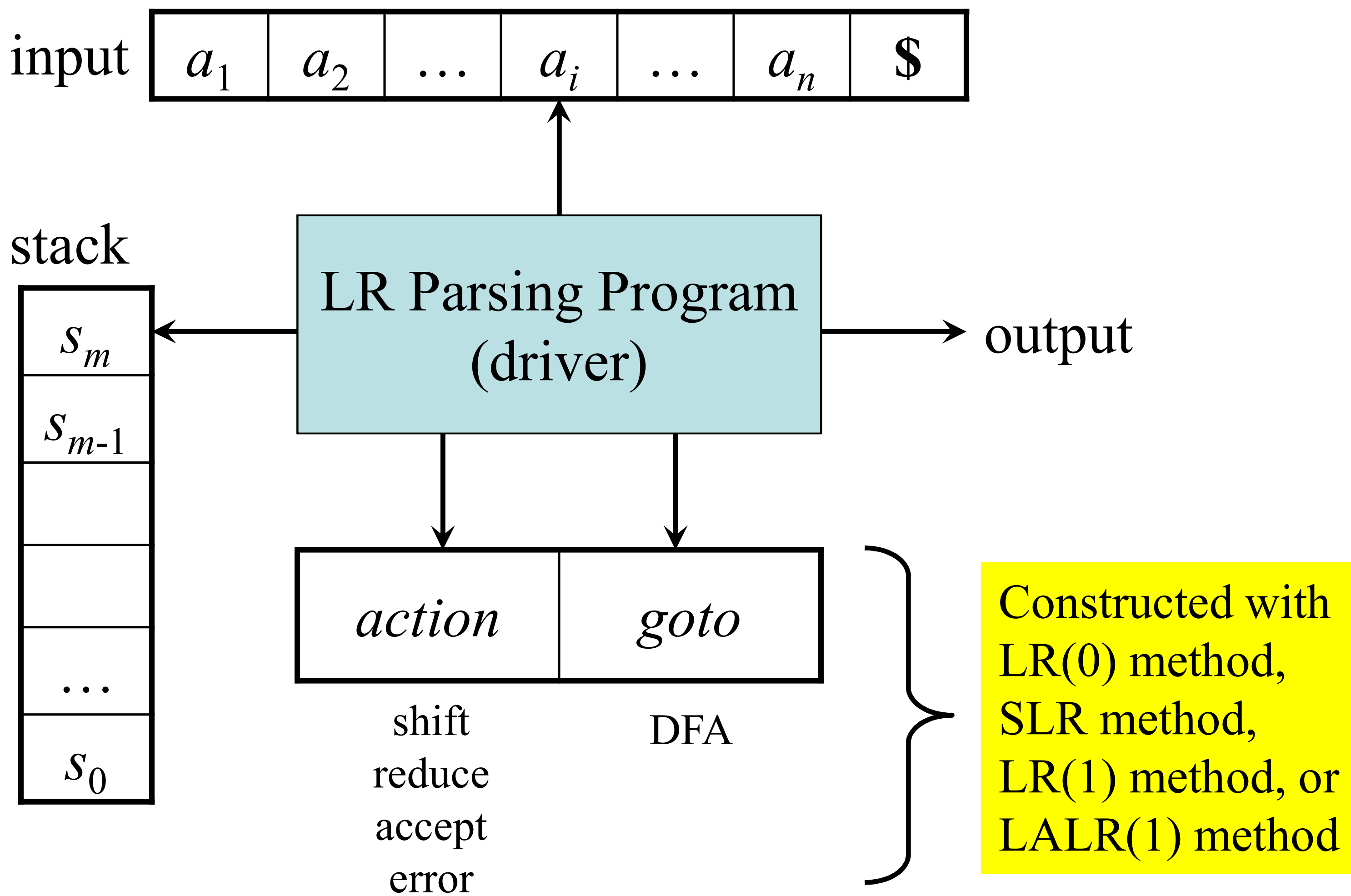
$A \to \mathbf{a}$

$B \to \mathbf{a}$

Stack	Symbols	Input	Action
0	**$**	**aa$**	shift to 3
0 3	**$a**	**a$**	reduce $A \to \mathbf{a}$
0 2	$A	**a$**	shift to 5
0 2 5	$A**a**	**$**	reduce $B \to \mathbf{a}$
0 2 4	$AB	**$**	reduce $C \to AB$
0 1	$C	**$**	accept ($S \to C$)



Model of an LR Parser



LR Parsing (Driver)

$X_1\, X_2 \ldots X_m\, a_i\, a_{i+1} \ldots a_n$ ← right-sentential form

Configuration (= LR parser state):

$$(\underbrace{s_0\, s_1\, s_2 \ldots s_m}_{stack},\ \underbrace{a_i\, a_{i+1} \ldots a_n\, \mathbf{\$}}_{input})$$

If $action[s_m,a_i]$ = shift s **then** push s, and advance input:

$$(s_0\, s_1 s_2 \ldots s_m\, s,\quad a_{i+1} \ldots a_n\, \mathbf{\$})$$

If $action[s_m,a_i]$ = reduce $A \rightarrow \beta$ and $goto[s_{m-r},A] = s$ with $r=|\beta|$ **then** pop r symbols, and push s:

$$(s_0\, s_1\, s_2 \ldots s_{m-r}\, s,\quad a_i\, a_{i+1} \ldots a_n\, \mathbf{\$})$$

If $action[s_m,a_i]$ = accept **then** stop

If $action[s_m,a_i]$ = error **then** attempt recovery

Example LR(0) Parsing Table

State I_0:
$C' \to \bullet C$
$C \to \bullet A\ B$
$A \to \bullet \mathbf{a}$

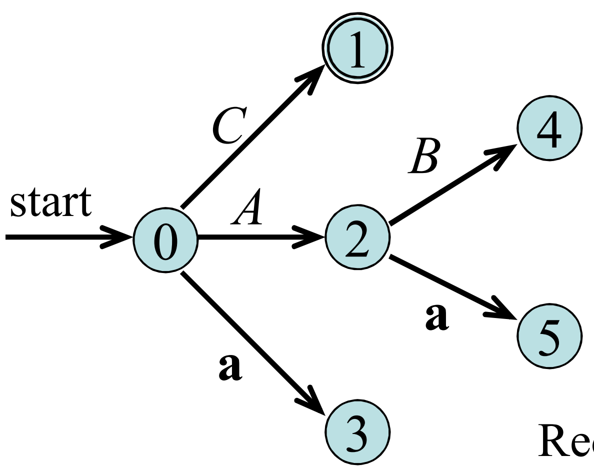
State I_1:
$C' \to C\bullet$

State I_2:
$C \to A \bullet B$
$B \to \bullet \mathbf{a}$

State I_3:
$A \to \mathbf{a}\bullet$

State I_4:
$C \to A\ B\bullet$

State I_5:
$B \to \mathbf{a}\bullet$



Shift & goto 3

Reduce by production #2

	action		*goto*		
state	**a**	**$**	*C*	*A*	*B*
0	s3		1	2	
1		acc			
2	s5				4
3	r3	r3			
4	r2	r2			
5	r4	r4			

Grammar:
1. $C' \to C$
2. $C \to A\ B$
3. $A \to \mathbf{a}$
4. $B \to \mathbf{a}$

SLR Grammars

- SLR (Simple LR): SLR is a simple extension of LR(0) shift-reduce parsing
- SLR eliminates some conflicts by populating the parsing table with reductions $A \to \alpha$ on symbols in FOLLOW(A)

$S \to E$
$E \to \textbf{id} + E$
$E \to \textbf{id}$

State I_0:
$S \to \bullet E$
$E \to \bullet\textbf{id} + E$
$E \to \bullet\textbf{id}$

$\xrightarrow{goto(I_0,\textbf{id})}$

State I_2:
$E \to \textbf{id}\bullet + E$
$E \to \textbf{id}\bullet$

$\xrightarrow{goto(I_2,+)}$

Shift on +

FOLLOW(E)={**$**}
thus reduce on **$**

SLR Parsing Table

- Reductions do not fill entire rows
- Otherwise the same as LR(0)

1. $S \rightarrow E$
2. $E \rightarrow \mathbf{id} + E$
3. $E \rightarrow \mathbf{id}$

	id	+	**$**	*E*
0	s2			1
1			acc	
2		s3	r3	
3	s2			4
4			r2	

Shift on +

FOLLOW(E)={**$**}
thus reduce on **$**

State I_1:
$S \rightarrow E\ \bullet$

State I_3:
$E \rightarrow \mathbf{id} + \bullet\ E$

State I_4:
$E \rightarrow \mathbf{id} + E\ \bullet$

SLR Parsing

- An LR(0) state is a set of LR(0) items
- An LR(0) item is a production with a • (dot) in the right-hand side
- Build the LR(0) DFA by
  - *Closure operation* to construct LR(0) items
  - *Goto operation* to determine transitions
- Construct the SLR parsing table from the DFA
- LR parser program uses the SLR parsing table to determine shift/reduce operations

Constructing SLR Parsing Tables

1. Augment the grammar with $S' \to S$
2. Construct $C=\{I_0, I_1, \ldots, I_n\}$, the collection of sets of *LR*(0) *items*. State i is constructed from I_i.
3. If $[A\to\alpha\bullet a\beta] \in I_i$ and ***goto***$(I_i, a)=I_j$ then set ***action***$[i, a]$=shift j, where a is a terminal
4. If $[A\to\alpha\bullet] \in I_i$ then set ***action***$[i, a]$=reduce $A\to\alpha$ for all a $\in$ FOLLOW(A) (apply only if $A\neq S'$)
5. If $[S' \to S\bullet]$ is in I_i then set ***action***$[i,$ **\$**$]$=accept
6. If ***goto***$(I_i, A)=I_j$ then set ***goto***$[i,A]=j$
7. Repeat 3-6 until no more entries added
8. The initial state i is the I_i holding item $[S' \to\bullet S]$

Example Grammar and LR(0) Items

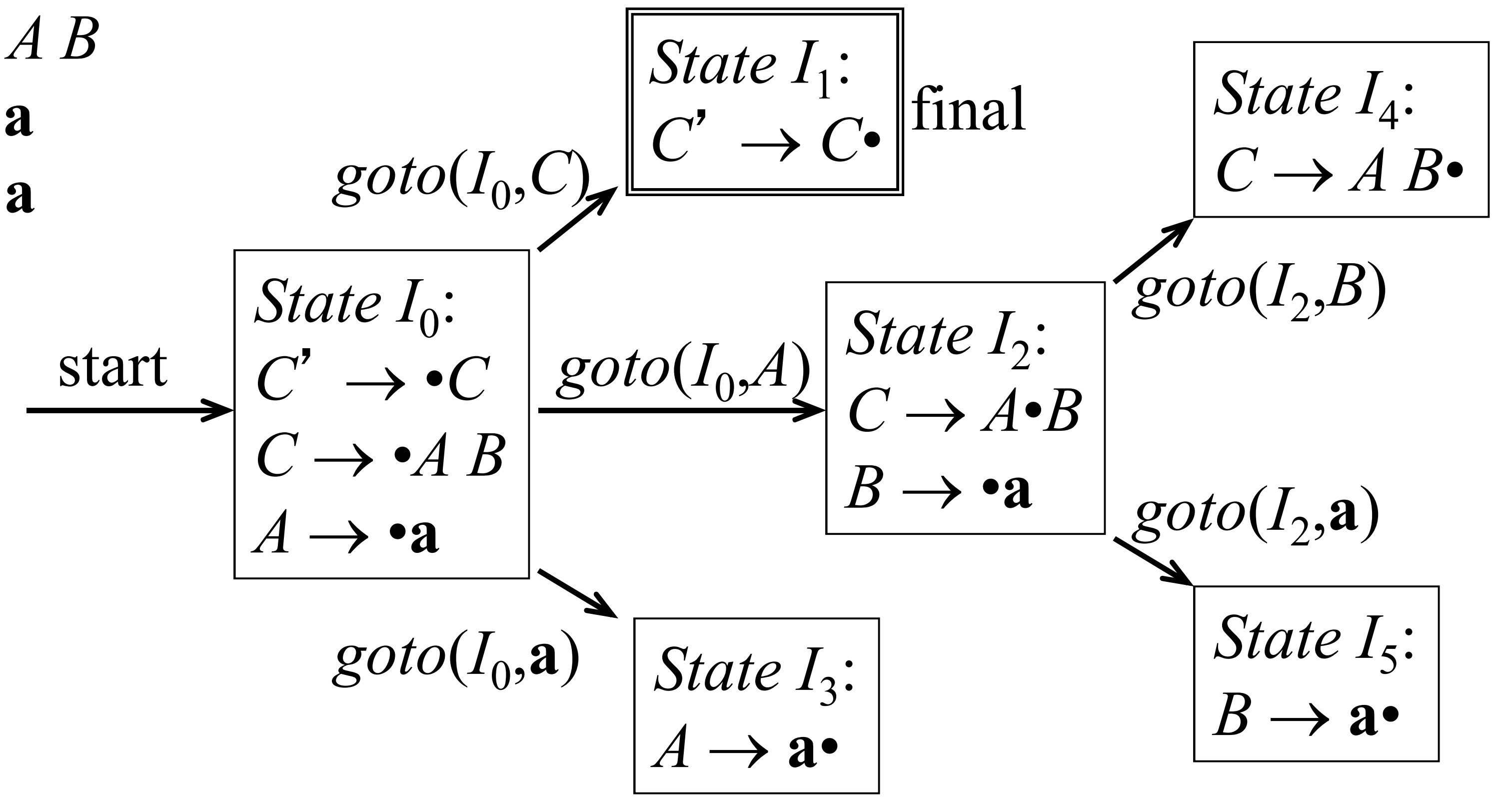
Augmented grammar:

1. $C' \to C$
2. $C \to A\ B$
3. $A \to \mathbf{a}$
4. $B \to \mathbf{a}$

$I_0 = closure(\{[C' \to \bullet C]\})$

$I_1 = \text{goto}(I_0, C) = closure(\{[C' \to C\bullet]\})$

…



Example SLR Parsing Table

State I_0:
$C' \to \bullet C$
$C \to \bullet A\ B$
$A \to \bullet\mathbf{a}$

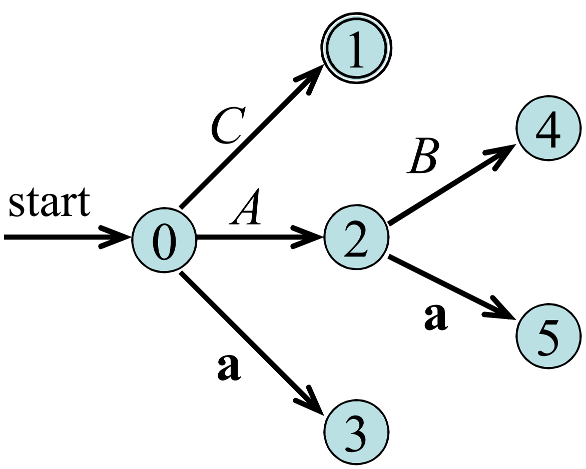
State I_1:
$C' \to C\bullet$

State I_2:
$C \to A\bullet B$
$B \to \bullet\mathbf{a}$

State I_3:
$A \to \mathbf{a}\bullet$

State I_4:
$C \to A\ B\bullet$

State I_5:
$B \to \mathbf{a}\bullet$

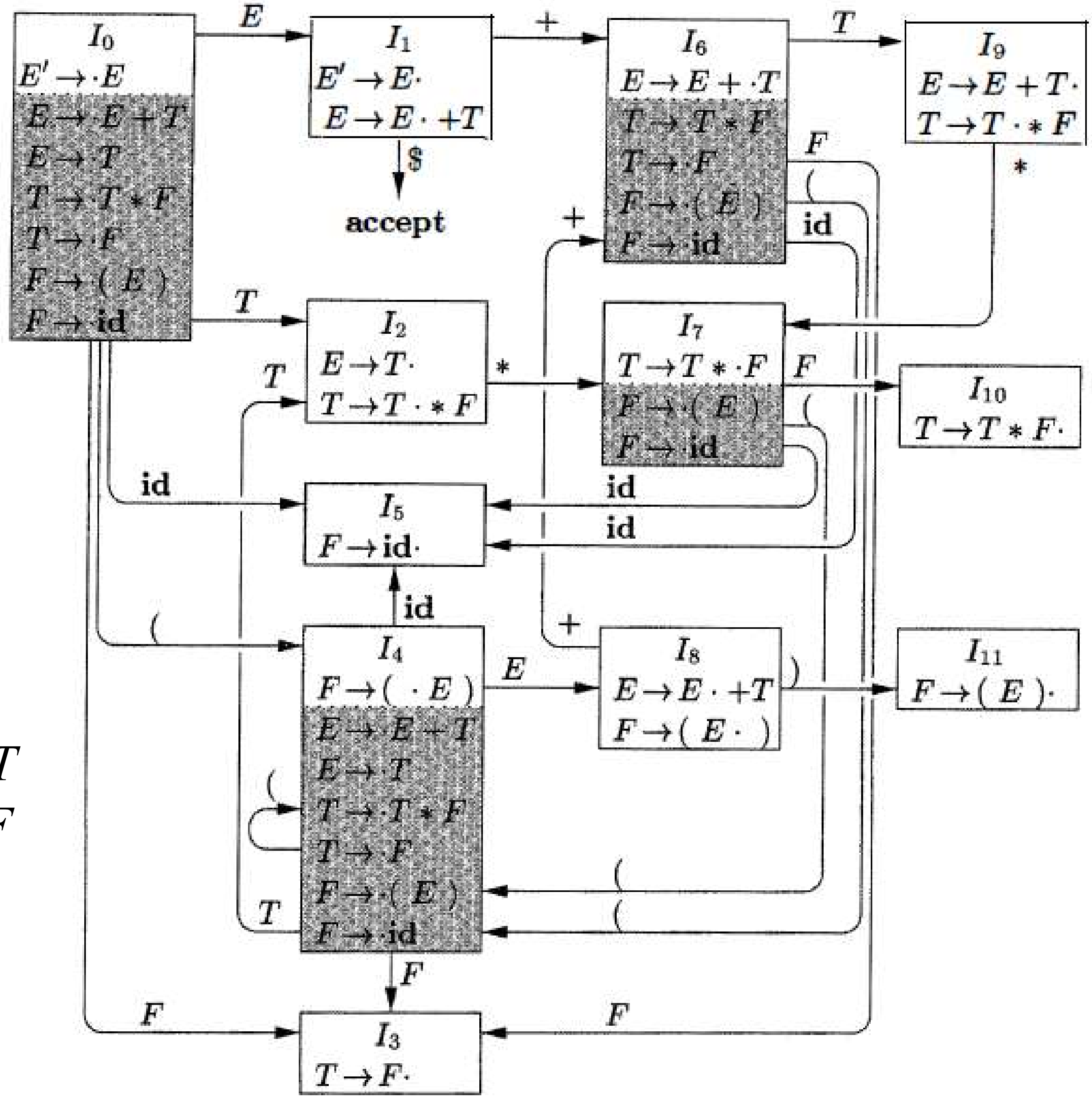


	action		*goto*		
state	**a**	**$**	*C*	*A*	*B*
0	s3		1	2	
1		acc			
2	s5				4
3	r3				
4		r2			
5		r4			

FOLLOW(A) = {**a**}
FOLLOW(C) = {**$**}
FOLLOW(B) = {**$**}

Grammar:
1. $C' \to C$
2. $C \to A\ B$
3. $A \to \mathbf{a}$
4. $B \to \mathbf{a}$

LR(0) Automaton for



Grammar:

$E \to E + T \mid T$

$T \to T * F \mid F$

$F \to (E)$

$F \to \mathbf{id}$

Another Example SLR Parse Table

Grammar:

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow$ **(** E **)**
6. $F \rightarrow$ **id**

	action						*goto*		
state	**id**	+	*	**(**	**)**	**$**	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Shift & goto 5

Reduce by production #1

Moves of an SLR parser on id * id + id

Using the SLR Parse Table on Previous Slide

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		$\mathbf{id} * \mathbf{id} + \mathbf{id}\,\$$	shift
(2)	0 5	$\mathbf{id}$	$* \mathbf{id} + \mathbf{id}\,\$$	reduce by $F \rightarrow \mathbf{id}$
(3)	0 3	F	$* \mathbf{id} + \mathbf{id}\,\$$	reduce by $T \rightarrow F$
(4)	0 2	T	$* \mathbf{id} + \mathbf{id}\,\$$	shift
(5)	0 2 7	$T *$	$\mathbf{id} + \mathbf{id}\,\$$	shift
(6)	0 2 7 5	$T * \mathbf{id}$	$+ \mathbf{id}\,\$$	reduce by $F \rightarrow \mathbf{id}$
(7)	0 2 7 10	$T * F$	$+ \mathbf{id}\,\$$	reduce by $T \rightarrow T * F$
(8)	0 2	T	$+ \mathbf{id}\,\$$	reduce by $E \rightarrow T$
(9)	0 1	E	$+ \mathbf{id}\,\$$	shift
(10)	0 1 6	$E +$	$\mathbf{id}\,\$$	shift
(11)	0 1 6 5	$E + \mathbf{id}$	$\$$	reduce by $F \rightarrow \mathbf{id}$
(12)	0 1 6 3	$E + F$	$\$$	reduce by $T \rightarrow F$
(13)	0 1 6 9	$E + T$	$\$$	reduce by $E \rightarrow E + T$
(14)	0 1	E	$\$$	accept

SLR, Ambiguity, and Conflicts

- SLR grammars are unambiguous
- But **not** every unambiguous grammar is SLR
- Consider for example the unambiguous grammar

1. $S \to L = R$ 2. $S \to R$
3. $L \to * R$ 4. $L \to \mathbf{id}$
5. $R \to L$

I_0:
$S' \to \bullet S$
$S \to \bullet L{=}R$
$S \to \bullet R$
$L \to \bullet{*}R$
$L \to \bullet\mathbf{id}$
$R \to \bullet L$

I_1:
$S' \to S\bullet$

I_2:
$S \to L\bullet{=}R$
$R \to L\bullet$

I_3:
$S \to R\bullet$

I_4:
$L \to {*}\bullet R$
$R \to \bullet L$
$L \to \bullet{*}R$
$L \to \bullet\mathbf{id}$

I_5:
$L \to \mathbf{id}\bullet$

I_6:
$S \to L{=}\bullet R$
$R \to \bullet L$
$L \to \bullet{*}R$
$L \to \bullet\mathbf{id}$

I_7:
$L \to {*}R\bullet$

I_8:
$R \to L\bullet$

I_9:
$S \to L{=}R\bullet$

action[2,=]=s6
action[2,=]=r5

no

Conflict: has no SLR parsing table!

Viable Prefixes

- During the LR parsing, the stack contents must be a prefix of a right-sentential form
  - If the stack holds α, the rest of input is *x*
  - There is a right-most derivation $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha x$
- But, not all prefixes of right-sentential forms can appear on the stack
  - The parser must not shift past the handle
  - Example: Suppose $E \overset{*}{\underset{rm}{\Rightarrow}} F * \mathbf{id} \underset{rm}{\Rightarrow} (E) * \mathbf{id}$,

    the stack must not hold $(E)*$, as (E) is a handle.
- The prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes

Viable Prefixes (Cont.)

- A viable prefix is a prefix of a right-sentential form that does not continue past the right end of the rightmost handle of that sentential form
- We say item A $\rightarrow \beta_1 \bullet \beta_2$ is valid for a viable prefix $\alpha\beta_1$ if there is a derivation $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta_1 \bullet \beta_2\, w$.
- A $\rightarrow \beta_1 \bullet \beta_2$ is valid for $\alpha\beta_1$ and $\alpha\beta_1$ is on the parsing stack
  - If $\beta_2 \neq \varepsilon$, then shift
  - $\beta_2 = \varepsilon$, then reduce

Viable Prefixes (Cont.)

- The set of valid items for a viable prefix δ is exactly the set of items reached from the initial state along the path labeled δ in the LR(0) automaton for the grammar
- Example: See state 7 of automaton on slide 16.

  $T \rightarrow T{*}\bullet F$, $F \rightarrow \bullet(E)$, and $F \rightarrow \bullet\mathbf{id}$ are valid items for E+T∗

$$E' \underset{rm}{\Rightarrow} E \underset{rm}{\Rightarrow} E+T \underset{rm}{\Rightarrow} E+T*F$$

$$E' \underset{rm}{\Rightarrow} E \underset{rm}{\Rightarrow} E+T \underset{rm}{\Rightarrow} E+T*F \underset{rm}{\Rightarrow} E+T*(E)$$

$$E' \underset{rm}{\Rightarrow} E \underset{rm}{\Rightarrow} E+T \underset{rm}{\Rightarrow} E+T*F \underset{rm}{\Rightarrow} E+T*\mathbf{id}$$

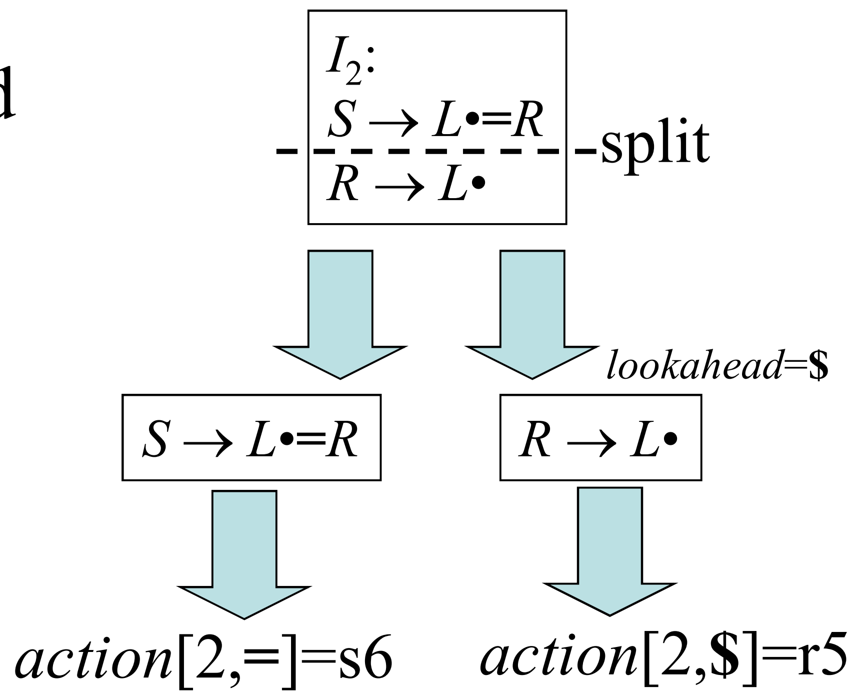
7. LR(1) Grammars

- SLR too simple
- LR(1) parsing uses lookahead to avoid unnecessary conflicts in parsing table
- LR(1) item = LR(0) item + lookahead

LR(0) item:	LR(1) item:
$[A \rightarrow \alpha \bullet \beta]$	$[A \rightarrow \alpha \bullet \beta, a]$

SLR Versus LR(1)

- Split the SLR states by adding LR(1) lookahead
- Unambiguous grammar
  1. $S \to L = R$
  2. $S \to R$
  3. $L \to * R$
  4. $L \to \mathbf{id}$
  5. $R \to L$



Should not reduce on =, because no right-sentential form begins with R=

LR(1) Items

- An *LR*(1) *item*

  $[A \to \alpha \bullet \beta, a]$

  contains a *lookahead* terminal a, meaning α already on top of the stack, expect to parse βa
- For items of the form

  $[A \to \alpha \bullet, a]$

  the lookahead a is used to reduce $A \to \alpha$ only if the next lookahead of the input is a
- For items of the form

  $[A \to \alpha \bullet \beta, a]$

  with $\beta \neq \varepsilon$ the lookahead has no effect

The Closure Operation for LR(1) Items

1. Start with *closure*(I) = I
2. If $[A \rightarrow \alpha \bullet B\beta, a] \in$ *closure*(I) then for each production $B \rightarrow \gamma$ in the grammar and each terminal $b \in \mathrm{FIRST}(\beta a)$, add the item $[B \rightarrow \bullet\gamma, b]$ to *closure*(I) if not already in *closure*(I)
3. Repeat 2 until no new items can be added

The Goto Operation for LR(1) Items

1. For each item $[A \rightarrow \alpha \bullet X\beta, a] \in I$, add the set of items $closure(\{[A \rightarrow \alpha X \bullet \beta, a]\})$ to $goto(I,X)$ if not already there
2. Repeat step 1 until no more items can be added to $goto(I,X)$

Constructing the set of LR(1) Items of a Grammar

1. Augment the grammar with a new start symbol S' and production $S' \rightarrow S$
2. Initially, set $C = \{ \mathit{closure}(\{[S' \rightarrow \bullet S, \$]\}) \}$ (this is the start state of the DFA)
3. For each set of items $I \in C$ and each grammar symbol $X \in (N \cup T)$ such that $\mathit{goto}(I,X) \notin C$ and $\mathit{goto}(I,X) \neq \emptyset$, add the set of items $\mathit{goto}(I,X)$ to C
4. Repeat 3 until no more sets can be added to C

Example Grammar and LR(1) Items

- Augmented LR(1) grammar (4.55):

  $S' \to S$

  $S \to C\ C$

  $C \to$ c C | d

- LR(1) items

$I_0: \quad S \to \cdot S, \$$
$\quad\quad S \to \cdot CC, \ \$$
$\quad\quad C \to \cdot cC, \ c/d$
$\quad\quad C \to \cdot d, \ c/d$

$I_1: \quad S' \to S\cdot, \ \$$

$I_2: \quad S \to C\cdot C, \ \$$
$\quad\quad C \to \cdot cC, \ \$$
$\quad\quad C \to \cdot d, \ \$$

$I_3: \quad C \to c\cdot C, \ c/d$
$\quad\quad C \to \cdot cC, \ c/d$
$\quad\quad C \to \cdot d, \ c/d$

$I_4: \quad C \to d\cdot, \ c/d$

$I_5: \quad S \to CC\cdot, \$$

$I_6: \quad C \to c\cdot C, \ \$$
$\quad\quad C \to \cdot cC, \ \$$
$\quad\quad C \to \cdot d, \ \$$

$I_7: \quad C \to d\cdot, \ \$$

$I_8: \quad C \to cC\cdot, \ c/d$

$I_9: \quad C \to cC\cdot, \ \$$

LR(1) items and goto Operation for Grammar (4.55)

I_0: $S \to \cdot S, \$$ goto(I_0, S) = I_1
$S \to \cdot CC,\ \$$ goto(I_0, C) = I_2
$C \to \cdot cC,\ c/d$ goto(I_0, c) = I_3
$C \to \cdot d,\ c/d$ goto(I_0, d) = I_4

I_1: $S' \to S\cdot,\ \$$

I_2: $S \to C\cdot C,\ \$$ goto(I_2, C) = I_5
$C \to \cdot cC,\ \$$ goto(I_2, c) = I_6
$C \to \cdot d,\ \$$ goto(I_2, d) = I_7

I_3: $C \to c\cdot C,\ c/d$ goto(I_3, C) = I_8
$C \to \cdot cC,\ c/d$ goto(I_3, c) = I_3
$C \to \cdot d,\ c/d$ goto(I_3, d) = I_4

I_4: $C \to d\cdot,\ c/d$

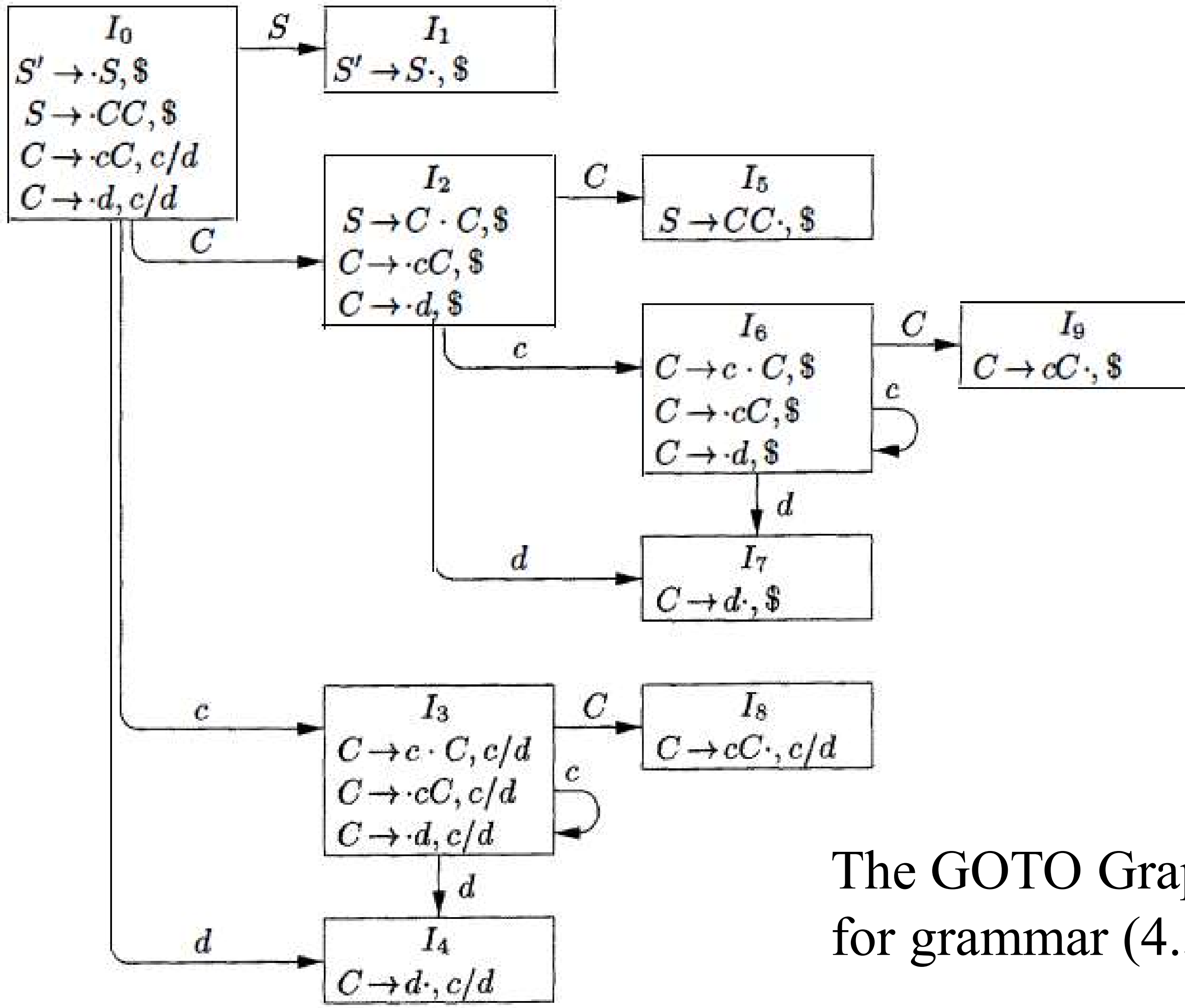
I_5: $S \to CC\cdot, \$$

I_6: $C \to c\cdot C,\ \$$ goto(I_6, C) = I_9
$C \to \cdot cC,\ \$$ goto(I_6, c) = I_6
$C \to \cdot d,\ \$$ goto(I_6, d) = I_7

I_7: $C \to d\cdot,\ \$$

I_8: $C \to cC\cdot,\ c/d$

I_9: $C \to cC\cdot,\ \$$



The GOTO Graph
for grammar (4.55)

Example Grammar and LR(1) Items

- Unambiguous LR(1) grammar:

  $S \rightarrow L = R$

  $S \rightarrow R$

  $L \rightarrow * \ R$

  $L \rightarrow \textbf{id}$

  $R \rightarrow L$

- Augment with $S' \rightarrow S$
- LR(1) items (next slide)

I_0:	$[S' \to \bullet S,\ \mathbf{\$}]$	goto(I_0,S)=I_1
	$[S \to \bullet L{=}R,\ \mathbf{\$}]$	goto(I_0,L)=I_2
	$[S \to \bullet R,\ \mathbf{\$}]$	goto(I_0,R)=I_3
	$[L \to \bullet{*}R,\ {=}/\mathbf{\$}]$	goto(I_0,*)=I_4
	$[L \to \bullet\mathbf{id},\ {=}/\mathbf{\$}]$	goto(I_0,**id**)=I_5
	$[R \to \bullet L,\ \mathbf{\$}]$	
I_1:	$[S' \to S\bullet,\ \mathbf{\$}]$	
I_2:	$[S \to L\bullet{=}R,\ \mathbf{\$}]$	goto(I_2,=)=I_6
	$[R \to L\bullet,\ \mathbf{\$}]$	
I_3:	$[S \to R\bullet,\ \mathbf{\$}]$	
I_4:	$[L \to {*}\bullet R,\ {=}/\mathbf{\$}]$	goto(I_4,R)=I_7
	$[R \to \bullet L,\ {=}/\mathbf{\$}]$	goto(I_4,L)=I_8
	$[L \to \bullet{*}R,\ {=}/\mathbf{\$}]$	goto(I_4,*)=I_4
	$[L \to \bullet\mathbf{id},\ {=}/\mathbf{\$}]$	goto(I_4,**id**)=I_5
I_5:	$[L \to \mathbf{id}\bullet,\ {=}/\mathbf{\$}]$	
I_6:	$[S \to L{=}\bullet R,\ \mathbf{\$}]$	goto(I_6,R)=I_9
	$[R \to \bullet L,\ \mathbf{\$}]$	goto(I_6,L)=I_{10}
	$[L \to \bullet{*}R,\ \mathbf{\$}]$	goto(I_6,*)=I_{11}
	$[L \to \bullet\mathbf{id},\ \mathbf{\$}]$	goto(I_6,**id**)=I_{12}
I_7:	$[L \to {*}R\bullet,\ {=}/\mathbf{\$}]$	
I_8:	$[R \to L\bullet,\ {=}/\mathbf{\$}]$	
I_9:	$[S \to L{=}R\bullet,\ \mathbf{\$}]$	
I_{10}:	$[R \to L\bullet,\ \mathbf{\$}]$	
I_{11}:	$[L \to {*}\bullet R,\ \mathbf{\$}]$	goto(I_{11},R)=I_{13}
	$[R \to \bullet L,\ \mathbf{\$}]$	goto(I_{11},L)=I_{10}
	$[L \to \bullet{*}R,\ \mathbf{\$}]$	goto(I_{11},*)=I_{11}
	$[L \to \bullet\mathbf{id},\ \mathbf{\$}]$	goto(I_{11},**id**)=I_{12}
I_{12}:	$[L \to \mathbf{id}\bullet,\ \mathbf{\$}]$	
I_{13}:	$[L \to {*}R\bullet,\ \mathbf{\$}]$	

Constructing Canonical LR(1) Parsing Tables

1. Augment the grammar with $S' \rightarrow S$
2. Construct the set $C=\{I_0,I_1,\ldots,I_n\}$ of LR(1) items
3. If $[A\rightarrow\alpha\bullet a\beta,\ b] \in I_i$ and $goto(I_i,a)=I_j$ then set $action[i,a]$=shift j
4. If $[A\rightarrow\alpha\bullet,\ a] \in I_i$ then set $action[i,a]$=reduce $\mathrm{A}\rightarrow\alpha$ (apply only if $A\neq S'$)
5. If $[S' \rightarrow S\bullet,\ \$]$ is in I_i then set $action[i,\$]$=accept
6. If $goto(I_i,A)=I_j$ then set $goto[i,A]=j$
7. Repeat 3-6 until no more entries added
8. The initial state i is the I_i holding item $[S' \rightarrow\bullet S,\$]$

Example Canonical LR(1) Parsing Table

Grammar:
0. $S' \rightarrow S$
1. $S \rightarrow C\ C$
2. $C \rightarrow \boldsymbol{c}\ C$
3. $C \rightarrow \boldsymbol{d}$

STATE	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Example LR(1) Parsing Table

Grammar:

1. $S' \rightarrow S$
2. $S \rightarrow L = R$
3. $S \rightarrow R$
4. $L \rightarrow * R$
5. $L \rightarrow \textbf{id}$
6. $R \rightarrow L$

	id	*	=	$	*S*	*L*	*R*
0	s5	s4			1	2	3
1				acc			
2			s6	r6			
3				r3			
4	s5	s4				8	7
5			r5	r5			
6	s12	s11				10	9
7			r4	r4			
8			r6	r6			
9				r2			
10				r6			
11	s12	s11				10	13
12				r5			
13				r4			

LALR Parsing

- LR(1) parsing tables have many states
- LALR parsing (Look-Ahead LR) merges two or more LR(1) state into one state to reduce table size
- Less powerful than LR(1)
  - Will not introduce shift-reduce conflicts, because shifts do not use lookaheads
  - May introduce reduce-reduce conflicts, but seldom do so for grammars of programming languages

Constructing LALR Parsing Tables

1. Construct sets of LR(1) items
2. Combine LR(1) sets with sets of items that share the same first part

Grammar:
0. S' → *S*
1. *S* → *C C*
2. *C* → ***c*** C
3. *C* → ***d***

I_{36}: $C \to c{\cdot}C,\ c/d/\$$
$C \to {\cdot}cC,\ c/d/\$$
$C \to {\cdot}d,\ c/d/\$$

I_{47}: $C \to d{\cdot},\ c/d/\$$

I_{89}: $C \to cC{\cdot},\ c/d/\$$

STATE	ACTION			GOTO	
	c	*d*	$	*S*	*C*
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Constructing LALR Parsing Tables

1. Construct sets of LR(1) items
2. Combine LR(1) sets with sets of items that share the same first part

$$\left.\begin{array}{ll}
I_4: & [L \to *\bullet R,\ \ =/\$] \\
 & [R \to \bullet L,\ \ =/\$] \\
 & [L \to \bullet *R,\ \ =/\$] \\
 & [L \to \bullet \mathbf{id},\ \ =/\$] \\
 & \\
I_{11}: & [L \to *\bullet R,\ \ \$] \\
 & [R \to \bullet L,\ \ \$] \\
 & [L \to \bullet *R,\ \ \$] \\
 & [L \to \bullet \mathbf{id},\ \ \$]
\end{array}\right\}
\begin{array}{l}
[L \to *\bullet R,\ \ =/\$] \\
[R \to \bullet L,\ \ =/\$] \\
[L \to \bullet *R,\ \ =/\$] \\
[L \to \bullet \mathbf{id},\ \ =/\$]
\end{array}$$

Shorthand for two items in the same set

Example Grammar and LALR Parsing Table

- Unambiguous LR(1) grammar:

$$
\begin{aligned}
S &\rightarrow L = R \\
&\;\mid R \\
L &\rightarrow * R \\
&\;\mid \textbf{id} \\
R &\rightarrow L
\end{aligned}
$$

- Augment with $S' \rightarrow S$
- LALR items (next slide)

I_0: $[S' \to \bullet S,\ \mathbf{\$}]$ goto(I_0,S)=I_1
$[S \to \bullet L{=}R,\ \mathbf{\$}]$ goto(I_0,L)=I_2
$[S \to \bullet R,\ \mathbf{\$}]$ goto(I_0,R)=I_3
$[L \to \bullet{*}R,\ {=}/\mathbf{\$}]$ goto(I_0,*)=I_4
$[L \to \bullet\mathbf{id},\ {=}/\mathbf{\$}]$ goto(I_0,**id**)=I_5
$[R \to \bullet L,\ \mathbf{\$}]$

I_1: $[S' \to S\bullet,\ \mathbf{\$}]$ goto(I_1,**$**)=acc

I_2: $[S \to L\bullet{=}R,\ {=}]$ goto(I_2,=)=I_6
$[R \to L\bullet,\ \mathbf{\$}]$

I_3: $[S \to R\bullet,\ \mathbf{\$}]$

I_4: $[L \to {*}\bullet R,\ {=}/\mathbf{\$}]$ goto(I_4,R)=I_7
$[R \to \bullet L,\ {=}/\mathbf{\$}]$ goto(I_4,L)=I_9
$[L \to \bullet{*}R,\ {=}/\mathbf{\$}]$ goto(I_4,*)=I_4
$[L \to \bullet\mathbf{id},\ {=}/\mathbf{\$}]$ goto(I_4,**id**)=I_5

I_5: $[L \to \mathbf{id}\bullet,\ {=}/\mathbf{\$}]$

I_6: $[S \to L{=}\bullet R,\ \mathbf{\$}]$ goto(I_6,R)=I_8
$[R \to \bullet L,\ \mathbf{\$}]$ goto(I_6,L)=I_9
$[L \to \bullet{*}R,\ \mathbf{\$}]$ goto(I_6,*)=I_4
$[L \to \bullet\mathbf{id},\ \mathbf{\$}]$ goto(I_6,**id**)=I_5

I_7: $[L \to {*}R\bullet,\ {=}/\mathbf{\$}]$

I_8: $[S \to L{=}R\bullet,\ \mathbf{\$}]$

I_9: $[R \to L\bullet,\ {=}/\mathbf{\$}]$

Shorthand for two items

$[R \to L\bullet,\ {=}]$
$[R \to L\bullet,\ \mathbf{\$}]$

Example LALR Parsing Table

Grammar:

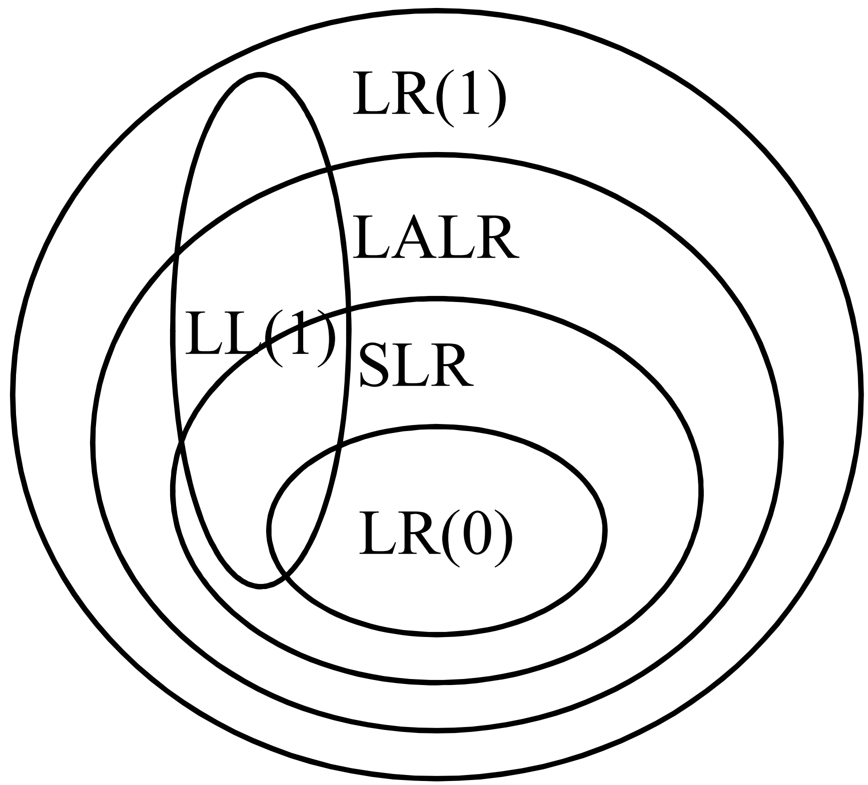
1. $S' \rightarrow S$
2. $S \rightarrow L = R$
3. $S \rightarrow R$
4. $L \rightarrow$ ***** R
5. $L \rightarrow$ **id**
6. $R \rightarrow L$

	id	*****	**=**	**$**	*S*	*L*	*R*
0	s5	s4			1	2	3
1				acc			
2			s6	r6			
3				r3			
4	s5	s4				9	7
5			r5	r5			
6	s5	s4				9	8
7			r4	r4			
8				r2			
9			r6	r6			

LL, SLR, LR, LALR Summary

- LL parse tables computed using FIRST/FOLLOW
  – Nonterminals × terminals → productions
  – Computed using FIRST/FOLLOW
- LR parsing tables computed using closure/goto
  – LR states × terminals → shift/reduce actions
  – LR states × nonterminals → goto state transitions
- A grammar is
  – LL(1) if its LL(1) parse table has no conflicts
  – SLR if its SLR parse table has no conflicts
  – LALR if its LALR parse table has no conflicts
  – LR(1) if its LR(1) parse table has no conflicts

LL, SLR, LR, LALR Grammars



8. Dealing with Ambiguous Grammars

1. $S' \rightarrow E$
2. $E \rightarrow E + E$
3. $E \rightarrow$ **id**

	id	+	**$**	E
0	s2			1
1		s3	acc	
2		r3	r3	
3	s2			4
4		s3/r2	r2	

stack	symbols	input
0	**$**	**id+id+id$**
…		…
0 1 3 4	**$**$E$+$E$	**+id$**

Shift/reduce conflict:
action[4,+] = shift 4
action[4,+] = reduce $E \rightarrow E + E$

When shifting on +:
yields right associativity
id+(id+id)

When reducing on +:
yields left associativity
(id+id)+id

Using Associativity and Precedence to Resolve Conflicts

- Left-associative operators: reduce
- Right-associative operators: shift
- Operator of higher precedence on stack: reduce
- Operator of lower precedence on stack: shift

$$S' \to E$$
$$E \to E + E$$
$$E \to E * E$$
$$E \to \textbf{id}$$

stack	symbols	input	
0	\$	**id*id+id\$**	
…		…	
0 1 3 5	\$$E$*$E$	**+id\$**	reduce $E \to E*E$

Error Detection in LR Parsing

- An LR parser will detect an error when it consults the parsing action table and finds an error entry.
- Canonical LR parser uses full LR(1) parse tables and will never make a single reduction before recognizing the error when a syntax error occurs on the input
- SLR and LALR may still reduce when a syntax error occurs on the input, but will never shift the erroneous input symbol

Error Recovery in LR Parsing

- Panic mode
  - Pop until state with a goto on a nonterminal *A* is found, (where *A* represents a major programming construct), push *A*
  - Discard input symbols until one is found in the FOLLOW set of *A*
- Phrase-level recovery
  - Implement error routines for every error entry in table
- Error productions
  - Pop until state has error production, then shift on stack
  - Discard input until symbol is encountered that allows parsing to continue